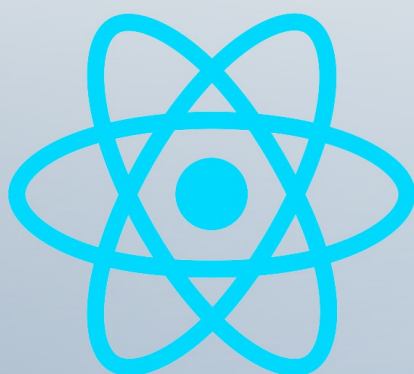


# Manual de React JS



Miguel Angel Alvarez  
Miguel Angel Durán  
David García



[desarrolloweb.com/manuales/manual-de-react.html](http://desarrolloweb.com/manuales/manual-de-react.html)

# Introducción: Manual de React

En este manual vamos a comenzar a explicar la librería React, una herramienta estupenda para el desarrollo de interfaces de usuario y aplicaciones Javascript del lado del cliente.

React fue creada por Facebook y actualmente es un software libre que se usa en cientos de páginas y aplicaciones de primer nivel, evolución natural para quienes usan otras librerías como jQuery.

En este manual iremos publicando los artículos sobre React, comenzando por una introducción general y una guía para los primeros pasos.

Encuentras este manual online en:

<http://desarrolloweb.com/manuales/manual-de-react.html>

## Autores del manual

Las siguientes personas han participado como autores escribiendo artículos de este manual.

---

### Miguel Angel Alvarez

Miguel es fundador de DesarrolloWeb.com y la plataforma de formación online EscuelaIT. Comenzó en el mundo del desarrollo web en el año 1997, transformando su hobby en su trabajo.



### Miguel Angel Durán

Desarrollador apasionado de la programación, con tecnologías fullstack desde PHP a NodeJS. Su Javascript del lado del cliente lo prefiere con la librería React.



### David García

David García es Front-End web developer en Schibsted Spain y lleva ya más de diez años desarrollando proyectos web en agencias como HTTPcomunicació y Runroom o en plataformas de e-commerce como LetsBonus.

# Qué es React y por qué interesa usarlo

## **Qué es React y los motivos por los que es una estupenda alternativa para el desarrollo de interfaces de usuario o aplicaciones web del lado del cliente.**

React es una librería Javascript focalizada en el desarrollo de interfaces de usuario. Esa es su principal área de trabajo, pero lo cierto es que con todo el ecosistema de aplicaciones y herramientas y componentes, con React encontramos un excelente aliado para hacer todo tipo de aplicaciones web, SPA (Single Page Application) o incluso aplicaciones para móviles.

Es por tanto una base sobre la cual se puede construir casi cualquier cosa con Javascript y que nos facilita mucho el desarrollo, ya que nos ofrece muchas cosas ya listas, en las que no necesitamos invertir tiempo para desarrollar. En este artículo te ampliaremos esta información, aportando además diversos motivos por los que elegir React como librería del lado del cliente.



## **Un poco de historia**

React es una librería que proviene de Facebook. Es software libre y a partir de su liberación una creciente comunidad de desarrolladores la está usando. Se crea en base a unas necesidades, generadas por el propio desarrollo de la web de la popular red social. En Facebook necesitaban herramientas para un desarrollo rápido pero focalizadas en un mayor rendimiento que otras alternativas existentes en el mercado.

Detectaron que el típico marco de binding y doble binding ralentizaba un poco su aplicación, debido a la cantidad de conexiones entre las vistas y los datos. Como respuesta crearon una nueva dinámica de funcionamiento, en la que optimizaron la forma como las vistas se renderizaban frente al cambio en los datos de la aplicación.

A partir de ahí la probaron en su red social con resultados positivos y luego en Instagram, también propiedad de Facebook, con éxito. Más adelante, alentados por los positivos resultados en el rendimiento de React, muchas otras aplicaciones web de primer nivel la fueron adoptando. BBC, Airbnb, Netflix, Dropbox y un largo etc.

## Cuál es el objetivo de React

---

Sirve para desarrollar aplicaciones web de una manera más ordenada y con menos código que si usas Javascript puro o librerías como jQuery centradas en la manipulación del DOM. Permite que las vistas se asocien con los datos, de modo que si cambian los datos, también cambian las vistas.

El código spaghetti que se suele producir mediante librerías como jQuery se pretende arquitecturizar y el modo de conseguirlo es a través de componentes. Una interfaz de usuario es básicamente creada a partir de un componente, el cual encapsula el funcionamiento y la presentación. Unos componentes se basan además en otros para solucionar necesidades más complejas en aplicaciones. También permite crear otras piezas de aplicación cómodamente, como los test.

## Comparación de React con otras librerías o frameworks

---

Con respecto a librerías sencillas como jQuery, React aporta una serie de posibilidades muy importante. Al tener las vistas asociadas a los datos, no necesitamos escribir código para manipular la página cuando los datos cambian. Esta parte en librerías sencillas es muy laboriosa de conseguir y es algo que React hace automáticamente.

También en comparación con jQuery nos permite una arquitectura de desarrollo más avanzada, con diversos beneficios como la encapsulación del código en componentes, que nos ofrecen una serie de ventajas más importantes que los plugin, como la posibilidad de que esos componentes conversen e interaccionen entre si, algo que sería muy difícil de conseguir con Plugins.

ReactJS solapa por completo las funcionalidades de jQuery, por lo que resulta una evolución natural para todos los sitios que usan esa librería. Podrían convivir pero no es algo que realmente sea necesario y recargaría un poco la página, por lo que tampoco sería muy recomendable.

Ya luego en comparación con frameworks como es el caso de Angular o Ember, React se queda a mitad de camino. Pero ojo, a partir de todo el ecosistema de React se llega más o menos a las mismas funcionalidades, así que es una alternativa perfecta.

Decimos que se queda a mitad de camino porque React por sí mismo es una librería y no un framework, puesto que React se ocupa de las interfaces de usuario. Quizás nos sirva decir que sería la "V" en un framework "MVC", aunque es solo una manera de hablar, puesto que React podría ocupar también parcelas de lo que sería la "C". Todo depende de nuestra manera de trabajar aunque, no obstante, esta posible carencia con respecto a los frameworks Javascript se soluciona con capas adicionales a React. Lo que podría interpretarse como una desventaja, muchos desarrolladores lo entienden como una ventaja con respecto a frameworks completos, ya que tú puedes desarrollar con React a tu gusto, aplicando aquellas herramientas y librerías adicionales que hacen las cosas como mejor se adapte al proyecto.

No se puede decir de una manera objetiva si es ReactJS es mejor o peor que otras alternativas, porque eso ya entra más en el terreno de la opinión. Lo cierto es que muchas librerías se

especializan en el "data-binding", pero React toma esa misma necesidad y la resuelve de otra manera. La diferencia es que React le pone más inteligencia a la necesidad de actualizar una vista cuando es necesario y lo consigue mediante el "DOM Virtual".

En líneas generales el virtual DOM es una representación del DOM pero en memoria. Cuando se actualiza una vista se actualiza el DOM Virtual, que es mucho más rápido que actualizar el DOM del navegador. Cuando React compara el DOM Virtual con el DOM del navegador sabe perfectamente qué partes de la página debe actualizar y se ahorra la necesidad de actualizar la vista entera. Es algo muy potente y es la base del rendimiento optimizado de React. Esto se hace de manera transparente para el desarrollador, que no necesita intervenir en nada.

## React es isomórfico

---

Éste es un concepto relativamente nuevo, pero muy interesante en el desarrollo de aplicaciones que se desean tengan un buen posicionamiento en buscadores. Básicamente se trata de que, con un mismo código, se pueda renderizar HTML tanto en el servidor como en el cliente, rebajando la carga de trabajo necesaria para realizar aplicaciones web amigables para buscadores.

El problema de las aplicaciones Javascript es que muchas veces reciben los datos en crudo del servidor, o un API, en formato JSON. Las librerías Javascript y frameworks toman esos datos para producir el HTML que debe representar el navegador. Esto, que es la solución más adecuada, porque nos permite desacoplar el servidor del cliente, pero se convierte en un aspecto negativo de cara al posicionamiento en buscadores como Google, debido a que el cuerpo de la página no tiene contenido.

**Nota:** Al no tener contenido una página que recibe los datos en un JSON, Google no sabe qué palabras clave son interesantes y no otorga ranking para ellas. Con ello la aplicación o página no consigue posicionar. Google está haciendo cambios y ha comenzado a procesar el Javascript para saber los datos de una página, pero aún dista de las ventajas que supone que el contenido esté en el propio HTML que entrega el servidor.

Para la librería que nos ocupa, React, permite isomorfismo. Eso es que, con el mismo código somos capaces de renderizar tanto en el cliente como el servidor. Por tanto, cuando llega un buscador como Google, con la misma base de código se le puede entregar el HTML con el contenido ya renderizado, lo que lleva a que una aplicación React sea capaz de posicionarse tan bien como una aplicación web tradicional que renderice del lado del servidor, como es el caso de un desarrollo tradicional o un desarrollo basado en un CMS como WordPress.

Todo esto se consigue gracias a NodeJS y se puede reutilizar no solo la parte de la presentación, sino también la lógica de negocio. En resumen React permite isomorfismo, algo que le faltaba tradicionalmente a AngularJS 1.x. Ahora se ha resuelto en Angular 2.x. Aunque muchas librerías siguen sin ser capaces de soportar isomorfismo.

## Ecosistema de React

---

Como hemos dicho, React en sí es una librería y como tal hay cosas que se deja del lado de fuera con respecto a soluciones aportadas por los frameworks MVC. Sin embargo existe todo un ecosistema de herramientas, aplicaciones y librerías que al final equiparan React a un framework.

Hay herramientas que se usan en múltiples proyectos, como el caso de Redux o Flux, que aportan partes que React no se encarga. Éstos se ocupan del flujo de datos en React y lo resuelven de una manera optimizada, elegante, poniendo énfasis en la claridad de las aplicaciones. Como desarrolladores podemos escoger entre varios frameworks encargados del flujo de los datos, basados en React. Como otros ejemplos tenemos generadores de aplicaciones, sistemas de routing del lado del cliente, etc.

Por otra parte, al desarrollar en base a componentes reutilizables permite que puedas usar el desarrollo de un proyecto en otro. Y por el mismo motivo, encuentras una amplia comunidad que libera sus propios componentes para que cualquier persona los pueda usar en cualquier proyecto. Por tanto, antes de desarrollar algo en React conviene ver si otro desarrollador ya ha publicado un componente que lo haga y en la mayoría de los casos, cuando se trata de cosas de ámbito general, veremos que siempre es así.

Hay componentes desde simples botones, sliders, tooltips, etc. Es muy sencillo que se pueda compartir, gracias a que los componentes son capaces de trabajar de manera independiente y que encapsulan funcionalidad para que no interaccionen con otros componentes si no se desea.

React Native es otra de las herramientas disponibles en el ecosistema, que permite llevar una aplicación escrita con Javascript y React como aplicación nativa para dispositivos iOS, Android, etc. Y se trata de aplicaciones nativas, no de web views.

## Conclusión

---

React es una librería completa y con un punto de maduración óptimo para implementarla en muchos tipos de proyectos distintos. Nos permite un desarrollo ágil, ordenado y con una arquitectura mantenible, focalizada en un gran performance.

Aunque React no se encarga de todas las partes necesarias para hacer una aplicación web compleja, la serie de componentes y herramientas diversas que se basan en React nos permite encontrar una alternativa capaz de hacer cualquier cosa que podríamos hacer con un complejo framework.

Todo esto te lo vamos a explicar en el Curso de desarrollo de aplicaciones con React, de EscuelaIT. Es una excelente alternativa para comenzar con React de la mano de expertos desarrolladores que ya están usando la librería desde hace tiempo.

Agradecemos a nuestros compañeros Gorka Laucirica y Miguel Angel Durán, que en un evento en directo en DesarrolloWeb y EscuelaIT nos informaron sobre todas estas características de React.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 04/10/2016  
Disponible online en <http://desarrolloweb.com/articulos/que-es-react-motivos-uso.html>



# Primeros pasos con React

## **Primeros pasos con React, la librería Javascript para el desarrollo de interfaces de usuario, a través del paquete Create React App.**

ReactJS es una librería Javascript potente, focalizada en la sencillez y el rendimiento, usada en cientos de proyectos de primer nivel como Facebook, Netflix o Airbnb. Merece la pena conocerla puesto que se postula como una evolución natural para miles de desarrolladores que usan librerías más sencillas como jQuery. React en resumen aporta diversas ventajas, permitiendo escribir menos código y aumentando su facilidad de mantenimiento de las aplicaciones.

Introducciones aparte, en este artículo pretendemos presentar React de una manera más práctica y para ello vamos a utilizar un paquete muy interesante, llamado "Create React App", puesto a disposición por el propio Facebook, creador de ReactJS.



## **Qué es Create React App**

Generalmente cuando se va a construir un sitio o app web se tiene que lidiar con una serie de herramientas que forman parte del tooling de un frontend developer, como gestores de paquetes, de tareas, transpiladores, linters, builders, live reload, etc. Toda esta serie de herramientas pueden tener su complejidad si se quieren aprender con el suficiente detalle como para comenzar a usarlas en un proyecto, pero son esenciales para un buen workflow.

Por tanto, si queremos ser detallistas y proveernos de las herramientas necesarias para ser productivos y eficientes, se puede hacer difícil la puesta en marcha en un proyecto. Ahí es donde entra Create React App, ofreciéndonos todo lo necesario para comenzar una app, pero sin tener que perder tiempo configurando herramientas. Comenzaremos una app con un par de comandos sencillos, obteniendo muchos beneficios de desarrolladores avanzados.

Instalamos Create React App con el siguiente comando de npm:

```
npm install -g create-react-app
```

Una vez lo hemos instalado de manera global, nos metemos en la carpeta de nuestros proyectos y lanzamos el comando para comenzar una nueva aplicación:

```
create-react-app mi-app
```

Mediante el anterior comando se cargarán los archivos de un proyecto vacío y todas las dependencias de npm para poder contar con el tooling que hemos mencionado. Una vez terminado el proceso, que puede tardar un poco, podemos entrar dentro de la carpeta de nuestra nueva app.

```
cd my-app/
```

Y una vez dentro hacer que comience la magia con el comando:

```
npm start
```

Observarás que, una vez lanzas el comando para iniciar la app, se abre una página en tu navegador con un mensaje de bienvenida. Ese es el proyecto que acabamos de crear. No obstante, en el propio terminal nos indicarán la URL del servidor web donde está funcionando nuestra app, para cualquier referencia posterior. De manera predeterminada será el <http://localhost:3000/>, aunque el puerto podría cambiar si el 3000 está ocupado.

Create React App tiene otra serie de comandos interesantes, como por ejemplo para crear los archivos de nuestro proyecto, listos para producción.

```
npm run build
```

Para mayores informaciones consultar la página del repositorio en Github:

<https://github.com/facebookincubator/create-react-app>

## Carpetas de nuestra app React

El listado de nuestra app recién creada es bastante sencillo. Observarás que tenemos varias carpetas:

- **node\_modules**: con las dependencias npm del proyecto
- **public**: esta es la raíz de nuestro servidor donde se podrá encontrar el index.html, el archivo principal y el favicon.ico que sería el icono de la aplicación.
- **src**: aquí es donde vamos a trabajar principalmente para nuestro proyecto, donde vamos a colocar los archivos de nuestros componentes React.

Además encontrarás archivos sueltos como:

- **README.md** que es el readme de Create React App, con cantidad de información sobre el proyecto y las apps que se crean a partir de él.
- **package.json**, que contiene información del proyecto, así como enumera las dependencias de npm, tanto para desarrollo como para producción. Si conoces npm no necesitarás más explicaciones.
- **.gitignore** que es el típico archivo para decirle a git que ignore ciertas cosas del proyecto a la hora de controlar el versionado del código.

Para nuestros primeros pasos no necesitamos más que entrar en la carpeta src y empezar a editar su código. De entre todos los archivos que encuentras en la carpeta src por ahora nos vamos a quedar con uno en concreto, src/App.js, en el que se crea el componente principal de nuestra app.

**Nota:** no lo hemos comentado, pero React fue una de las primeras librerías que están orientadas a componentes. Con React debemos construir componentes que son como etiquetas nuevas de HTML que encapsulan su contenido, presentación y funcionalidad. A base de tener unos componentes que engloban a otros es como se consigue la arquitectura de componentes, que es la base del desarrollo más actual en las librerías y frameworks más destacados.

Si abres src/App.js verás que la mayoría del código que aparece es como si fuera HTML, aunque encerrado dentro de un script Javascript. Es la manera en la que ReactJS trabaja con las vistas de la aplicación. En realidad no es código HTML sino "JSX", una extensión a la sintaxis de Javascript que nos permite de una manera amigable editar y mantener el HTML que necesitas para "renderizar" (pintar) los componentes. JSX tiene muchas cosas interesantes que comentar, pero no vamos a entretenernos hasta más adelante.

## Hola Mundo en React

En el mencionado archivo src/App.js nos encontraremos poco código, pero para no despistarnos en nuestro primer "Hola Mundo", podemos borrar gran parte y quedarnos solamente con esto:

```
import React, { Component } from 'react';

class App extends Component {
  render() {
    return (
      <div className="App">
        <h1>Hola Mundo</h1>
      </div>
    );
  }
}

export default App;
```

En la primera línea "import React..." se está importando la librería React y la clase Component, que es la que usamos para, en la segunda línea, extender la clase App.

**Nota:** En React hay varias alternativas para crear componentes. Esta que se ha utilizado es la forma de ECMAScript 6, que sería la recomendada. No necesitas preocuparte porque ese Javascript no sea compatible con todos los navegadores, ya que el propio Create React App nos ha configurado ya el transpilador para convertirlo a Javascript compatible con ECMAScript 5.

Las clases que usamos para implementar componentes React solo necesitan un método render para poder funcionar. En nuestro caso observarás que el componente que estamos creando, llamado App (como el nombre de la clase), solamente tiene ese método render().

```
class App extends Component {  
  render() {  
    return (  
      <div className="App">  
        <h1>Hola Mundo</h1>  
      </div>  
    );  
  }  
}
```

A su vez, render devuelve un contenido que es el HTML necesario para pintar el componente en la página.

Guardando el archivo podrás observar como el navegador refresca la página automáticamente, gracias a las herramientas incluidas de serie dentro de Create React App.

## Arquitectura de componentes

Quizás sea un poco ambicioso comenzar ahora a hablar de arquitectura de componentes, ya que es un concepto muy amplio, pero nos viene bien para ver cómo se puede conseguir desarrollar una aplicación a base de usar varios componentes.

Vamos a mejorar un poquito nuestro ejemplo anterior y para ello, dentro del mismo App.js, vamos a crear un segundo componente para ver cómo unos componentes se pueden basar en otros. El código de nuestro nuevo componente será el siguiente:

```
class OtroSaludo extends Component {  
  render() {  
    return (  
      <p>Hola desde otro componente</p>  
    );  
  }  
}
```

```
}
```

Usar este componente nuevo es tan simple como colocar una etiqueta en el componente que lo necesite usar, que tendrá el mismo nombre que el de la clase que acabamos de crear para este componente.

```
<div className="App">
  <h1>Hola Mundo</h1>
  <OtroSaludo />
</div>
```

Debes observar cómo hemos usado la etiqueta "OtroSaludo" en la vista del componente App. Pondremos todo de momento dentro del mismo App.js, con lo que el código completo del archivo nos quedará como esto:

```
import React, { Component } from 'react';

class App extends Component {
  render() {
    return (
      <div className="App">
        <h1>Hola Mundo</h1>
        <OtroSaludo />
      </div>
    );
  }
}

class OtroSaludo extends Component {
  render() {
    return (
      <p>Hola desde otro componente</p>
    );
  }
}

export default App;
```

**Nota:** obviamente, podríamos (deberíamos) separar el código de cada componente en archivos distintos. No significa mucha dificultad, pero no vamos a entrar en ese detalle en este ejemplo.

En este momento nuestra aplicación te mostrará otro resultado cuando la veas en el navegador.

## Conclusión y vídeo

Esto es solo una pequeña muestra de lo que se puede hacer con un mínimo conocimiento de React. Cabe señalar que React es válido para hacer componentes que se pueden usar dentro de aplicaciones ya existentes, por ejemplo un proyecto PHP tradicional, que renderiza HTML del lado del servidor, como en aplicaciones enteramente del lado del cliente, que renderizan el HTML por medio de vistas React y con datos que traerás con JSON de un API REST o de cualquier otro medio.

Todo esto lo veremos en el futuro, pero si te interesa contar ya mismo con una formación completa y tutorizada para aprender React, te recomendamos participar en el [Curso de React de EscuelaIT](#).

Como presentación de estas funcionalidades básicas de React y otras que no hemos tratado en este artículo te recomendamos asistir al vídeo siguiente, de los primeros pasos en React.

Para ver este vídeo es necesario visitar el artículo original en:  
<http://desarrolloweb.com/articulos/primeros-pasos-react.html>

Este artículo es obra de *Miguel Angel Durán*  
Fue publicado por primera vez en 29/09/2016  
Disponible online en <http://desarrolloweb.com/articulos/primeros-pasos-react.html>

# Características de React

**En este artículo vamos a conocer algunas de las principales características de React.**

Antes de comenzar a programar nuevos ejemplos con React nos vamos a tomar nuevamente algo de tiempo para la teoría, en concreto para analizar la librería y así conocer más de cerca algunas de sus características. Ya tendremos tiempo en el [Manual de React](#) de ponernos algo más prácticos.

**React es una librería para crear aplicaciones web, con Javascript del lado del cliente y del lado del servidor.** Esa definición hace hincapié en una dualidad (cliente + servidor) que resulta una de las principales características de la librería, de las que ya hemos hablado anteriormente, en el artículo de [Qué es React y por qué usarlo](#). Aunque volveremos sobre ese punto, ahora vamos a conocer otros datos de interés.



## Composición de componentes

Así como en programación funcional se pasan funciones como parámetros para resolver problemas más complejos, creando lo que se conoce como composición funcional, en ReactJS podemos aplicar este mismo patrón mediante la composición de componentes

Las aplicaciones se realizan con la composición de varios componentes. Estos componentes encapsulan un comportamiento, una vista y un estado. Pueden ser muy complejos, pero es algo de lo que no necesitamos preocuparnos cuando estamos desarrollando la aplicación, porque el comportamiento queda dentro del componente y no necesitamos complicarnos por él una vez se ha realizado.

Esto es algo sobre lo que ya hemos hablado. En resumen, al desarrollar crearemos componentes para resolver pequeños problemas, que por ser pequeños son más fáciles de resolver y en adelante son más fáciles de visualizar y comprender. Luego, unos componentes se apoyarán en otros para resolver problemas mayores y al final la aplicación será un conjunto de componentes que trabajan entre sí. Este modelo de trabajo tiene varias ventajas, como la facilidad de mantenimiento, depuración, escalabilidad, etc.



## Desarrollo Declarativo Vs Imperativo

---

En la experiencia de desarrollo con librerías más sencillas como jQuery, o el propio "Vanilla Javascript" realizamos un estilo de programación imperativo. En ese estilo se realizan scripts que paso por paso tienen que informar sobre qué acciones o cambios en el DOM se deben realizar. Hay que ser muy concisos en esas acciones, especificando con detalle cada uno de los cambios que se quieren realizar. La forma imperativa de declarar nos obliga a escribir mucho código, porque cada pequeño cambio se debe definir en un script y cuando el cambio puede ser provocado desde muchos sitios, cuando agregamos eventos, el código comienza a ser poco mantenible.

Sin embargo, el estilo de React es más declarativo, en el que nosotros contamos con un estado de la aplicación y sus componentes reaccionan ante el cambio de ese estado. Los componentes tienen una funcionalidad dada y cuando cambia una de sus propiedades ellos producen un cambio. En el código de nuestra aplicación tendremos ese componente, y en él se declarará de donde vienen los datos que él necesita para realizar su comportamiento. Podremos usarlo tantas veces como queramos declarando que lo queremos usar y declarando también los datos que él necesita para funcionar.

Quizás ahora, si no tenemos experiencia con otras librerías o frameworks de enfoque declarativo la idea pueda quedar algo confusa, pero lo iremos viendo mejor con los ejemplos.

## Flujo de datos unidireccional

---

Ésta es otra de las cosas que facilita React, aunque no es exclusivo. En este modelo de funcionamiento, los componentes de orden superior propagan datos a los componentes de orden inferior. Los de orden inferior trabajarán con esos datos y cuando cambia su estado podrán propagar eventos hacia los componentes de orden superior para actualizar sus estados.

Este flujo tiende a ser unidireccional, pero entre componentes hermanos muchas veces se hace más cómodo que sea bidireccional y también se puede hacer dentro de React. Sin embargo, si tratamos siempre de mantener el patrón de funcionamiento unidireccional, nos facilitará mucho el mantenimiento de la aplicación y su depuración.

## Performance gracias al DOM Virtual

---

El desempeño de React es muy alto, gracias a su funcionamiento. Nos referimos al desempeño a la hora del renderizado de la aplicación. Esto se consigue por medio del DOM Virtual. No es que React no opere con el DOM real del navegador, pero sus operaciones las realiza antes sobre el DOM Virtual, que es mucho más rápido.

El DOM Virtual está cargado en memoria y gracias a la herramienta que diferenciación entre él y el real, el DOM del navegador se actualiza. El resultado es que estas operaciones permiten actualizaciones de hasta 60 frames por segundo, lo que producen aplicaciones muy fluidas, con movimientos suavizados.

## Isomorfismo

---



Es la capacidad de ejecutar el código tanto en el cliente como el servidor. También se conoce como "Javascript Universal". Sirve principalmente para solucionar problemas de posicionamiento tradicionales de las aplicaciones Javascript.

Como anteriormente ya hemos hablado sobre este punto, lo vamos a dejar por aquí.

## Elementos y JSX

---

ReactJS no retorna HTML. El código embebido dentro de Javascript, parece HTML pero realmente es JSX. Son como funciones Javascript, pero expresadas mediante una sintaxis propia de React llamada JSX. Lo que produce son elementos en memoria y no elementos del DOM tradicional, con lo cual las funciones no ocupan tiempo en producir pesados objetos del navegador sino simplemente elementos de un DOM virtual. Todo esto, como hemos dicho, es mucho más rápido.

React DOM y la herramienta de diffing, se encargarán más tarde de convertir esos elementos devueltos por JSX en DOM Real, sin que nosotros tengamos que intervenir.

## Componentes con y sin estado

---

React permite crear componentes de diversas maneras, pero hay una diferencia entre componentes con y sin estado.

Los componentes stateless son los componentes que no tienen estado, digamos que no guardan en su memoria datos. Eso no quiere decir que no puedan recibir valores de propiedades, pero esas propiedades siempre las llevarán a las vistas sin producir un estado dentro del componente. Estos componentes sin estado se pueden escribir con una sencilla función que retorna el JSX que el componente debe representar en la página.

Los componentes statefull son un poco más complejos, porque son capaces de guardar un estado y mantienen lógica de negocio generalmente. Su principal diferencia es que se escriben en el código de una manera más compleja, generalmente por medio de una clase ES6 (Javascript con ECMAScript 2015), en la que podemos tener atributos y métodos para realizar todo tipo de operaciones. Los componentes statefull, con estado, necesitan tener un método `render()` que se encarga de devolver el JSX que usar para representarlo en la página.

Podemos decir que los componentes con estado son componentes "listos" y los que no tienen estado son "tontos". Los statefull se usan para resolver problemas mayores, con lógica de negocio, mientras que los stateless se usan más para interfaces de usuario.

## Ciclo de vida de los componentes

---

React implementa un ciclo de vida para los componentes. Son métodos que se ejecutan cuando pasan cosas comunes con el componente, que nos permiten suscribir acciones cuando se produce una inicialización, se recibe la devolución de una promesa, etc.

En definitiva, todas estas cosas las iremos conociendo a lo largo del [Manual de React de Desarrolloweb.com](http://desarrolloweb.com/manuales/manual-de-react.html). Aunque recuerda que si quieres aprender mucho mejor, con menos tiempo y ayudado por profesores, tienes el [Curso de React de EscuelaIT](#).

Este artículo es obra de *David García*

Fue publicado por primera vez en 20/10/2016

Disponible online en <http://desarrolloweb.com/articulos/caracteristicas-react.html>

# Comenzar con React en un entorno vacío

**Práctica de un proyecto React básico y simplificado al máximo, mediante un entorno vacío, que comenzaremos con un archivo html en blanco.**

Vamos a crear un nuevo artículo para dar los primeros pasos con React, pero desde un entorno diferente. De hecho, usaremos un entorno vacío para comenzar, es decir, una página en blanco y sin la necesidad de descargar ninguna librería. Solo con un index.html y un poco de código.

Sin embargo, cabe aclarar que no sería la manera más adecuada para desarrollar React. ¿Entonces por qué explicamos cómo comenzar con React usando esta alternativa de desarrollo?. Para responder la pregunta aclararemos el objetivo del artículo: ofrecer una vía sencilla para poner en marcha React sin las complejidades de un entorno de desarrollo avanzado. Hay personas que prefieren enfrentarse a una nueva tecnología desde un lienzo vacío, o una página en blanco, donde tengamos pocas cosas y nos sean familiares.

**Nota:** Si no te asusta tener un entorno complejo y prefieres código ES6 te interesa mejor ver cómo comenzar React a través de Create React App. Esa aproximación es la que vimos en el artículo [Primeros pasos con React](#).

Además, si alguien ya tiene una estructura de aplicación y desea integrar React en ella, este artículo le dará una buena idea de cómo se podría comenzar. El problema en este caso es que React usa JSX, que requiere de un compilado por medio de Babel. A esta operación se la conoce como "transpilado", término a medio camino entre compilación y traducción. Por ese motivo al menos, para integrar Babel en un proyecto existente, estaríamos obligados a realizar una transpilación antes de poner en producción el código, lo que nos obligaría a definir nuestro entorno necesariamente.



## CDN de React

Comenzaremos entonces cargando React desde el CDN, de modo que no tengamos que instalar

en nuestro proyecto ninguna librería. React se ofrece con un CDN sobre el que podemos encontrar información y las versiones más actuales desde la [página de descargas](#).

El código que deberemos usar a partir del CDN está dividido en varios scripts. Usaremos los tres siguientes:

Por orden de aparición, estos scripts sirven para lo siguiente:

- react es la propia librería React
- react-dom nos permite comenzar con react usando el lenguaje JSX, para escribir algo parecido a HTML / XML cuando queremos definir la salida de la aplicación
- babel-core sirve para convertir el JSX en código Javascript compatible con todos los navegadores

Como recomendación es interesante que usemos, en la etapa de desarrollo, los archivos de React y React-DOM no minimizados. Así podrás obtener errores más claros cuando algo no funcione en tu página. Los anteriores scripts eran las versiones para desarrollo, pero en la página de descargas de React encontrarás esos mismos scripts minimizados y adecuados para entornos de producción.

**Nota:** Sobre JSX hemos hablado poco. Es un superset de Javascript que permite escribir código HTML dentro de un script Javascript. Es algo que se hace extraño al principio, pero que resulta muy potente. Puedes verlo inicialmente como un sistema de templating, donde de una manera clara y mantenible puedo generar salida en HTML, interpolando datos e incluso funcionalidad que viene de Javascript. Tampoco vamos a introducirnos ahora al detalle. Una de las grandes novedades de React en el panorama de desarrollo frontend es justamente el JSX.

## Anclaje de la salida de React

Ahora que ya tenemos nuestras tres librerías necesarias, podemos crear un script que use react para producir una pequeña salida. Sin embargo, necesitamos algún sitio de nuestro HTML donde volcar esta salida. Este sitio será una sencilla etiqueta HTML a la que le pondremos un identificador para poder referirnos a ella.

La salida con React será inyectada en esa etiqueta. Nosotros simplemente al ejecutar React especificaremos el id que tiene el elemento donde inyectar esa salida. React se encargará de lo demás.

El código de nuestra página HTML, una vez colocados los scripts del CDN y el elemento de anclaje quedaría de esta manera.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Test React desde una página en blanco</title>
```

```
</head>
<body>
  <div id="anclaje"></div>

  <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
  <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></script>
  <script src="https://unpkg.com/babel-core@5.8.38/browser.min.js"></script>
</body>
</html>
```

Como habrás observado, el elemento de anclaje es el DIV con id="anclaje". A continuación ya podremos poner en marcha React para hacer un sencillo "Hola mundo".

## Script React

Ahora vamos con lo que todos estarán esperando ver, cómo ejecutar React para producir esa salida, que se inyectará en el elemento de anclaje. Es bien sencillo, a través del objeto ReactDOM.

ReactDOM permite hacer cosas como renderizar HTML en un lugar de la página. La gracia es que ReactDOM soporta escribir código en JSX, de modo que nos simplifica escribir HTML dentro del propio Javascript. El método que usamos de ReactDOM es render(), el cual requiere dos parámetros: 1) El JSX con el HTML a generar. 2) el objeto del DOM donde inyectar la salida.

El ejemplo podría ser algo como esto, aunque luego veremos alguna variación:

```
<script type="text/babel">
  var contenido = <h1>Hola Mundo!!</h1>;
  var anclaje = document.getElementById('anclaje');
  ReactDOM.render(contenido, anclaje);
</script>
```

El método "render" de ReactDOM recibe dos parámetros. El contenido que debe renderizar y el objeto del DOM donde volcar el resultado de esa renderización. Esos dos parámetros son los que hemos generado en las variables contenido y anclaje generadas en las líneas anteriores.

Observarás, y seguramente te llamará la atención, la variable "contenido". A ella le hemos asignado algo que parece una cadena de texto con código HTML. Sin embargo, no es un string de código HTML (porque en ese caso debería ir entre comillas), sino código JSX. El JSX no se indica entre comillas, porque no es una cadena, sino código Javascript.

Como se ha dicho, conseguimos que el navegador entienda el código JSX gracias a la conversión de este script mediante babel-core. Es por ello que el script que estamos utilizando tiene un atributo type="text/babel", que indica que es un código que BabelJS debe procesar antes de que el navegador lo interprete.

**Nota:** Es muy importante el `type="text/babel"` porque si no lo pones el navegador te arrojará un error de sintaxis en el anterior código. Pero ojo, como hemos advertido antes, tampoco es la manera adecuada de trabajar con Babel, porque de este modo estaríamos poniendo la conversión del código del lado del cliente y eso es algo que se debe hacer en el proceso de despliegue. Si lo haces así como está en este ejemplo estarás obligando al navegador a dos cosas que producirán una aplicación poco optimizada: 1) descargar el código de Babel-core, con el peso que ello conlleva para la página y 2) procesar esa transpilación (traducción + compilación) del código JSX en tiempo de ejecución, haciendo el que procesado de la página sea más lento. Quizás no lo notes con una aplicación pequeña que se ejecuta en local, pero sería patente con una aplicación mayor ejecutada en remoto.

Como alternativa, veamos ahora el mismo código pero expresado de otra manera. En vez de generar variables en pasos anteriores enviamos los valores directamente al método `render()`.

```
<script type="text/babel">
  ReactDOM.render(
    <h1>Hola Mundo React</h1>,
    document.getElementById('example')
  )
</script>
```

## Ejemplo completo del hola mundo react sin setup del entorno de desarrollo

Ahora puedes ver el código de nuestra página, un archivo HTML que podrías poner en cualquier lugar de tu disco duro y ejecutar directamente con un doble clic. Sin dependencias de ningún otro paso que debas realizar para que esto funcione correctamente.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Test React desde una página en blanco</title>
</head>
<body>
  <div id="anclaje"></div>

  <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
  <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></script>
  <script src="https://unpkg.com/babel-core@5.8.38/browser.min.js"></script>
  <script type="text/babel">
    var contenido = <h1>Hola Mundo!!</h1>;
    var anclaje = document.getElementById('anclaje');
    ReactDOM.render(contenido, anclaje);
  </script>
</body>
</html>
```

Es un ejemplo de React básico, pero como hemos dicho poco apropiado por varios motivos.

Además, aunque el archivo se ejecute correctamente haciendo doble clic sobre él, te conviene acceder a los ejemplos a través de un servidor web, al menos ir acostumbrándote a ello, porque más adelante las cosas no funcionarán desde file:// y sí desde http://.

Espero que te haya parecido interesante esta práctica para comenzar con ReactJS sin tener que preocuparte de la creación de un entorno de desarrollo y una carpeta de aplicación.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 25/10/2016  
Disponible online en <http://desarrolloweb.com/articulos/comenzar-con-react-entorno-vacio.html>

# Componentes React con `createClass`

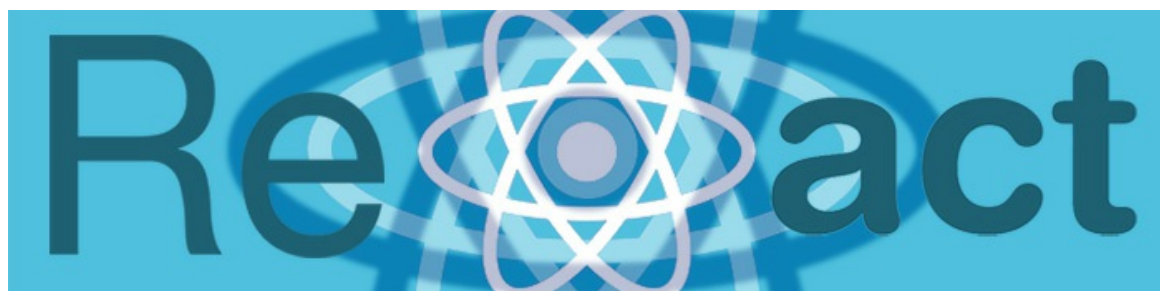
## Cómo crear componentes ReactJS con el método más tradicional, el método `createClass()` del objeto `React`.

Comenzamos un tema importante en React, ya que en esta librería realizamos un desarrollo basado en componentes. Básicamente quiere decir que, mediante anidación de componentes podremos crear aplicaciones completas de una manera modular, de fácil mantenimiento a pesar de poder ser complejas.

Como ya hemos comentado anteriormente sobre las bondades del desarrollo basado en componentes, vamos a abordar más técnicamente este apartado, observando algunas cosas interesantes sobre cómo debemos desarrollar componentes mediante la alternativa más clásica.

Los componentes se definen de diversas maneras. Lo ideal es que encontremos la que más se adapta a nuestros gustos o necesidades. Veremos en esta ocasión un método llamado `createClass()`, que es el más tradicional a la hora de construir componentes, y que nos permite empezar sin la necesidad de crear un entorno de desarrollo complejo para producir nuestras aplicaciones.

En futuros artículos estudiaremos alternativas que a nuestro juicio son más recomendables.



## Crear un componente con `createClass()`

Comencemos creando un componente por medio de un método que pertenece al objeto `React`, llamado `createClass()`. No es la manera más novedosa de crear componentes pero sí la más tradicional. Luego veremos cómo crear los componentes con clases de verdad, de las que nos permite realizar Javascript con ES6, pero `createClass()` es interesante de ver por dos motivos.

1. Porque nos permite crear componentes sin necesidad de crear un entorno de desarrollo complicado.
2. Porque hay muchos tutoriales más antiguos que éste donde podrás encontrar código como el que te vamos a proponer ahora. Así te será más fácil entender otras posibles



referencias de estudio.

**Nota:** Este modo de desarrollo haría continuación al artículo [Comenzar con React en un entorno vacío](#). Sería interesante echarle un vistazo a ese artículo para entender algunas cosas que ahora no vamos a explicar de nuevo.

En el código que encuentras a continuación observarás que para crear un componente usamos `React.createClass()`, enviando como parámetro la configuración del componente, indicada por medio de un literal de objeto Javascript.

```
var MiComponente = React.createClass({
  render: function() {
    return (
      <div>
        <h1>componente con createClass()</h1>
        <p>Este componente React lo creo como simple prueba. Lo hago con CreateClass porque quiero centrarme en ES5</p>
      </div>
    );
  }
});
```

Dentro del objeto de configuración enviado colocamos todas las cosas que debería tener nuestro componente. De momento observarás que le hemos pasado único método: `render()`, que es lo mínimo que debería tener todo componente básico.

El método `render()` contiene el código que debe usarse para representar el componente en la página. Como estamos dentro de React podemos usar la sintaxis JSX.

Como `render()` solamente sirve para devolver código que representar en la vista, lo único que encontramos es un `return`. Ya dentro del `return` estará el código JSX que construye la vista a usar para la representación del componente.

## Usar un componente

Un componente que hemos creado mediante `createClass()` no produce ninguna salida. Tendremos que usar el componente para que realmente podamos verlo en funcionamiento en la página. Esta parte requiere de la otra pieza de React que necesitamos para funcionar: `ReactDOM`.

**Nota:** Volvemos a referirnos al artículo de [hola mundo en React desde un archivo en blanco y entorno vacío](#), donde explicamos conceptos como `React-DOM` y el modo de usar estas piezas de React.

En este caso, como contenido a representar a `React-DOM`, le indicamos el código del

componente que acabamos de crear. Recuerda que como segundo parámetro tenemos que indicar un objeto del DOM donde anclar la visualización del componente.

```
ReactDOM.render(<MiComponente />, anclaje);
```

Una de las posibilidades de desarrollar con componentes es que, una vez definido el componente, lo podemos usar varias veces en el código de nuestra aplicación. Para ello simplemente colocamos varias veces su etiqueta. Su contenido entonces se renderizaría en cada sitio donde lo estamos usando.

Sin embargo, esto así no te funcionará:

```
ReactDOM.render(<MiComponente /><MiComponente />, anclaje);
```

El contenido del método `render()` no es posible escribirlo con un número de etiquetas hermanas. Si quieres usar cualquier número de veces ese componente tendrás que envolverlo en una etiqueta cualquiera. Esto es porque JSX es como si produjera un `return` por cada etiqueta que tiene y no es posible que una función devuelva dos `return` diferentes. La envoltura nos permite que solo se devuelva una única salida, aunque luego esa etiqueta pueda tener hijos, nietos, etc. en cualquier estructura de árbol.

Ahora con el siguiente código sí obtendremos resultados positivos.

```
ReactDOM.render(  
  <div>  
    <MiComponente />  
    <MiComponente />  
  </div>,  
  anclaje  
)
```

Observarás como hemos colocado unos saltos de línea en el código, simplemente para que se pueda leer mejor por el ojo humano. Estos saltos de línea son perfectamente válidos y dulcifican la sintaxis de JSX, a la vez que producen un código más mantenible y similar al HTML.

### Ejemplo completo componente react con `createClass()`

El ejemplo completo de este ejercicio lo tienes a continuación.

```
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    <meta charset="UTF-8">  
    <title>Test React desde una página en blanco</title>  
  </head>
```

```
<body>
  <div id="anclaje"></div>

  <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
  <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></script>
  <script src="https://unpkg.com/babel-core@5.8.38/browser.min.js"></script>
  <script type="text/babel">
var MiComponente = React.createClass({
  render: function() {
    return (
      <div>
        <h1>componente con createClass()</h1>
        <p>Este componente React lo creo como simple prueba. Lo hago con CreateClass porque quiero centrarme en ES5</p>
      </div>
    );
  }
});

ReactDOM.render(
  <div>
    <MiComponente />
    <MiComponente />
  </div>,
  anclaje
);

</script>
</body>
</html>
```

Ese código lo puedes ejecutar tal cual, copiando y pegando en cualquier fichero con extensión .html y luego abriéndolo en tu navegador. No necesita de nada más para funcionar, salvo que tengas conexión a Internet para que se puedan cargar los códigos de los scripts cargados desde el CDN de React.

De momento no hay mucho más que agregar, solo que permanezcas atento a futuros artículos donde estaremos explicando otros mecanismos interesantes que nos permiten crear componentes con clases ES6.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 09/11/2016  
Disponible online en <http://desarrolloweb.com/articulos/componentes-react-createclass.html>

# Componentes React mediante clases ES6

## Explicación y ejemplos sobre la alternativa de desarrollo de componentes React basada en la utilización de clases ES6.

En el pasado artículo del [Manual de React](#) comenzamos a explicar cómo crear componentes. En aquella ocasión lo hicimos con el método `createClass()` y ahora conoceremos otro mecanismo más actual, basado en clases de las que nos aporta ECMAScript 2015.

No es que por hacer los componentes mediante clases podamos realizar cosas adicionales a las que nos permitía el método de `createClass()`, sino que nos permitirá organizar el código de nuestras aplicaciones de una manera diferente, más práctica. Básicamente podremos colocar cada componente en su módulo independiente e importarlo cuando sea necesario con `imports` de ES6. Al mismo tiempo, el hecho de ser clases nos permitirá de beneficiarnos de los beneficios de la Programación Orientada a Objetos en general.



En la documentación de React afirman que los componentes con clases ES6 tienen un mejor desempeño que usar `createClass()`, que se notará en aplicaciones grandes. Como desventaja, tendremos que escribir algún código adicional que React ya te hace de manera automática si usas `createClass()`, como por ejemplo el bindeo del contexto (`this`) en los métodos definidos como manejadores de eventos. Hablaremos de ello con detalle llegado el momento.

Quizás pienses en que agregar ES6 te añade la dificultad en la operativa de desarrollo, por la necesidad de la transpilación. Pero dado que ya estamos en la necesidad de transpilar código JSX, será indiferente tener que transpilar también ES6.

**Nota:** Para ir ajustándonos a un proyecto real, en lo sucesivo trabajaremos sobre una instalación de Create React App, que nos facilita tener el entorno ya listo, con transpiladores, linters, live reload, etc. Recuerda que esto ya lo explicamos y lo probamos en el artículo de los [primeros pasos con ReactJS](#), así que esperamos que puedas despejar tus dudas allí.

## Cada componente en su archivo

Ahora que vamos a trabajar con ES6, por facilitarnos la vida en lo que respecta a la organización del código del proyecto, vamos a tener el cuidado de crear cada componente en su propio archivo. Cuanto menores los archivos más fáciles serán de mantener.

Cada archivo de código Javascript independiente forma parte de lo que se conoce como "módulo". Así que "cada componente en su módulo" sería más correcto de decir.

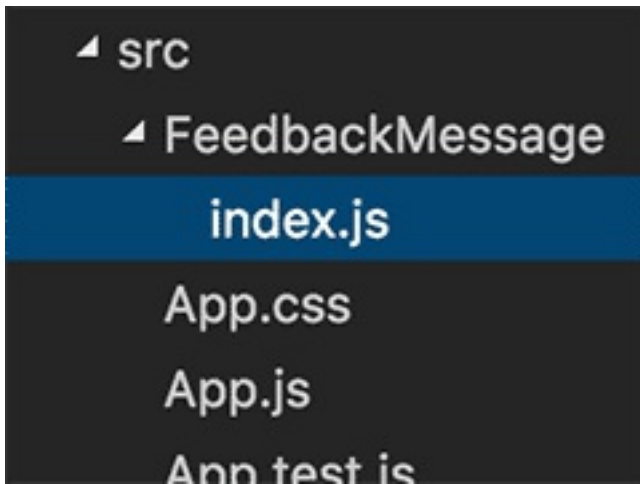
**Nota:** Seguimos la regla de hacer componentes pequeños, con una responsabilidad acotada, igual que en la programación orientada a objetos las clases deben tener una responsabilidad limitada. Si te preocupa que sean muchos archivos pequeños, y eso produzca un sitio poco optimizado, cabe decir que más adelante en el proceso de "build" se empaquetarán en un "bundle", que es un archivo compactado, que recoge el código de nuestra aplicación en un solo bloque. Esto es algo que hace Webpack, que es otra de las cosas que nos incluye Create React App.

Los archivos de los componentes los colocarás en la carpeta "src", contenida en un proyecto en blanco de Create React App. Ya luego cómo deseas organizar los códigos dentro de esa carpeta puede variar, pero una buena práctica sería crear una carpeta para cada componente o incluso carpetas de grupos de componentes que tengan una responsabilidad asociada. Iremos viendo estructuras más complejas en el futuro y también tus propios gustos y necesidades te irán marcando una pauta. Nosotros de momento nos contentaremos con colocar los archivos del componente en su carpeta independiente.

Así pues, dentro de src podrás crear una carpeta que se llame "MiComponente" o "mi-componente" o y dentro un "index.js" que tendrá el código del componente.

**Nota:** Yo personalmente como regla prefiero no usar nombres de archivos o carpetas en el disco duro con mayúsculas y separo las palabras con guiones. Pero en React la costumbre más habitual es colocar mayúsculas en los archivos, de modo que sean igual que el nombre de las clases. En realidad poco importa en la práctica, sólo acuérdate el nombre de tu carpeta para cuando la tengas que importar, en el momento que quieras usar ese componente.

Pienso que casi ni hace falta, pero en la siguiente imagen tenemos una estructura de carpetas y archivo que vamos a crear.



## Construir el componente, extendiendo de Component

Ahora dentro de nuestro index.js colocamos el código del componente. Comenzamos con la necesidad de importar React y la clase Component. React es la propia librería y Component es la clase de la que vamos a extender nuestros componentes.

```
import React, { Component } from 'react';
```

A continuación podemos hacer ya la clase ES6 en la que definimos el componente.

```
class FeedbackMessage extends Component {  
  render() {  
    return (  
      <div>  
        Bienvenido a FeedbackMessage!  
      </div>  
    )  
  }  
}
```

Simplemente fíjate que estamos extendiendo de Component. El resto del código ya lo debes de reconocer, con su método render que devuelve el HTML a renderizar. Siendo que lo que hay en el return no es HTML, sino JSX.

Una vez tenemos la clase, nos tenemos que preocupar de exportarla, para que se conozca desde fuera de este módulo. Esto se consigue con la sentencia "export" de ES6.

```
export default FeedbackMessage
```

Estamos diciendo que exporte la clase "FeedbackMessage", que es la clase con la que hemos implementado el componente. Además con "default" estamos permitiendo que no tengamos que especificar el nombre de la clase cuando queremos importarla, en el siguiente paso.

Solo un detalle, que podríamos definir el export al mismo tiempo que declaramos la clase, con una cabecera combinada como esta:

```
export default class FeedbackMessage extends Component {
```

## Usar un componente definido como clase

El único detalle nuevo ahora es que, para usar el componente, debemos importar su código definido en el módulo independiente. Lo conseguimos a partir de un "import" de ES6. Si estás familiarizado con esta práctica no tendrás duda alguna, pero vamos con ello.

```
import FeedbackMessage from './FeedbackMessage'
```

Ese import lo colocarás en cualquier módulo que desee usar tu componente nuevo. En el caso de un proyecto de Create React App tenemos el archivo App.js, que es donde se desarrolla todo el trabajo con la aplicación. Colocaremos allí el import de nuestro componente.

Una vez importado el componente lo podemos usar, ya dentro de un JSX. Usaremos el método render del componente App que hay en App.js. Simplificado al máximo, nos podrá quedar algo como esto.

```
class App extends Component {  
  render() {  
    return (  
      <FeedbackMessage />  
    );  
  }  
}
```

Ahora en nuestra App lo único que tenemos es el componente, pero como ya conoces un poco JSX sabes que podrás colocar otro tipo de HTML que sea necesario.

## Conclusión

Las clases ES6 nos facilitan muchas cosas en nuestro desarrollo y siempre es una buena idea tratar de aprovechar las ventajas de las versiones de Javascript más recientes. Para mantener una coherencia, vamos a intentar centrarnos en el manual en el desarrollo mediante clases ES6 en lo sucesivo.

Ten muy en cuenta Create React App, que te ofrece una estructura de proyecto con un entorno ya perfectamente configurado. También nuestros ejemplos los crearemos basando su funcionamiento en Create React App, porque así podemos abstraernos de varios aspectos de la configuración del proyecto. Recuerda que ya presentamos Create React App en el artículo de [Primeros pasos con React](#).

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 24/11/2016  
Disponible online en <http://desarrolloweb.com/articulos/componentes-react-clases-es6.html>



# Propiedades en componentes React

**Explicaciones sobre qué son las propiedades, en el desarrollo de componentes React. Veremos varios métodos de usar propiedades en componentes.**

Uno de los recursos elementales que tenemos al desarrollar componentes React son las propiedades. Nos sirven básicamente para cargar de datos a los componentes, de modo que éstos puedan personalizar su comportamiento, o para transferir datos de unos componentes a otros para producirse la interoperabilidad entre componentes. Existen diversas maneras de pasar propiedades, aunque lo más común es hacerlo de manera declarativa, al usar el componente dentro del código JSX.

Ya hemos visto en el [Manual de React](#) que existen diversas maneras de implementar componentes, y todavía veremos otras formas. En este caso es importante porque, dependiendo de cómo hemos definido los componentes en React, las propiedades nos llegarán por parámetros o como atributos de objeto. Analizaremos ambos casos en este artículo. Observarás que es muy sencillo compartir información y aprender a personalizar el comportamiento de los objetos.



## Propiedades en componentes definidos mediante clases

Comenzaremos observando cómo usar propiedades en componentes que hemos definido mediante clases de ES6, ya que es un conocimiento más fresco, que aprendimos justo en el anterior artículo, dedicado a la creación de componentes usando ECMAScript 2015.

Al usar clases, las propiedades enviadas al componente (enseguida mostraremos cómo usar el componente y enviarle las propiedades) las vamos a recibir en forma de propiedades o atributos del objeto. Por tanto accedemos a ellas a través de "this". En concreto usaremos this.props para acceder a un objeto donde tenemos todas las propiedades definidas como atributos.

Imaginemos que a nuestro componente le pasamos dos propiedades, llamadas "nombre" y "app". Entonces podremos usar esas propiedades de la siguiente manera:

```
import React, { Component } from 'react';

export default class FeedbackMessage extends Component {
  render() {
    return (
      <div>
        Bienvenido {this.props.nombre} a {this.props.app}
      </div>
    )
  }
}
```

El único detalle nuevo aquí es que estamos usando propiedades: `this.props.nombre` contendrá el valor pasado en la propiedad "nombre" y `this.props.app` el valor de la propiedad "app".

También es destacable el hecho de que nuestras propiedades se encuentran encerradas entre llaves. Las llaves son importantes, porque es la manera con la que se escapa un código JSX, permitiendo colocar dentro sentencias Javascript "nativo". Dicho de otra manera, al colocar las llaves estamos abriendo la posibilidad de escribir sentencias Javascript. Aquello que devuelvan esas sentencias se volcará como contenido en la vista.

## Cómo pasar propiedades a los componentes

A la hora de usar un componente podemos enviarle propiedades con sus valores. Es una operación que se hace comúnmente de forma declarativa en el código JSX, cuando usamos un componente.

Cada propiedad se indica como si fuera un atributo del HTML, al que le indicamos su valor. Este sería un código donde se usa el componente. Fíjate como los valores de mis propiedades son indicados como atributos del componente "nombre" y "app", que son las propiedades usadas en el ejemplo anterior.

```
import React, { Component } from 'react'
import FeedbackMessage from './FeedbackMessage'

class App extends Component {
  render() {
    return (
      <div className="App">
        <FeedbackMessage nombre="Miguel Angel Alvarez" app="Mi App React" />
      </div>
    );
  }
}
```

## Propiedades en componentes generados a partir de funciones

Otro modo de definir este tipo de componentes sencillos es por medio de simples funciones, lo

que dulcifica bastante la sintaxis de implementación de un componente. En este caso las propiedades se recibirán por parámetro en la función que implementa el componente.

Nota: Este tipo de componentes sencillos normalmente se conoce como "componentes sin estado". En un capítulo posterior veremos una contraposición entre los componentes con y sin estado.

```
import React from 'react'

export default function (propiedades){
  return (
    <div>
      <p>Soy una función!</p>
      <p>Hola {propiedades.nombre}, estás en {propiedades.app}</p>
    </div>
  )
}
```

Observarás que al hacer el import ya no necesitamos traernos la clase Component, sino solamente React.

Luego, al definir la función se prescinde del método render, porque no estamos haciendo una clase. La propia función es el equivalente al método render() que teníamos al crear componentes por medio de una clase ES6. Por lo tanto, devuelve el JSX para representar el componente.

En lo que respecta al uso de propiedades debes observar que te llegan mediante un parámetro.

```
export default function (propiedades){
```

Al parámetro le he llamado "propiedades", pero como es un simple parámetro tú podrás llamarlo como quieras. Algo más común sería usar el nombre "props" que es como React las nombra habitualmente. He usado una palabra en español justamente para remarcar que, al ser un parámetro, yo le puedo poner el nombre que desee.

Luego accederás a esas propiedades con el nombre que hayas puesto a tu parámetro, seguido de un punto y el nombre de la propiedad.

```
Hola {propiedades.nombre}, estás en {propiedades.app}
```

Como antes, para volcar datos que viene de Javascript dentro de un JSX, usamos la notación de las llaves.

Nota: El uso de un componente definido con una función no difiere del uso de cualquier otro componente, simplemente que debido a que estamos usando una función no podemos almacenar estado en este componente. Volveremos sobre esto del estado más adelante.

Ya que estamos en ES6, podríamos perfectamente usar las conocidas "arrow functions" para definir este componente. La sintaxis todavía quedará más resumida!

```
export default propiedades => (
  <div>
    <p>Soy una arrow function!</p>
    <p>Hola {propiedades.nombre}, estás en {propiedades.app}</p>
  </div>
)
```

## Propiedades de componentes creados con createClass()

Si utilizamos la alternativa tradicional de creación de componentes React con createClass(), el uso de propiedades también se hace por medio de la referencia "this.props".

```
var FeedbackMessage = React.createClass({
  render: function() {
    return (
      <div>
        <p>Soy un componente con createClass()</p>
        <p>También puedo usar propiedades {this.props.nombre} y {this.props.app}</p>
      </div>
    )
  }
})
```

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 09/12/2016  
Disponible online en <http://desarrolloweb.com/articulos/propiedades-componentes-react.html>

# El estado en los componentes React

**Qué son los componentes, con estado y sin estado, en React y cómo gestionar el estado de los componentes adecuadamente.**

En este artículo vamos a adentrarnos un poco más en el desarrollo de componentes React, dedicando nuestro estudio al estado de los componentes. Existen dos tipos de componentes, con estado (statefull components) y sin estado (stateless components). Los veremos en contraposición.

Todavía no hemos visto demasiados ejemplos, así que aprovecharemos también para ver algunos otros y así ir familiarizándonos con el desarrollo usando esta librería Javascript.

Recuerda que en pasados artículos hemos introducido el desarrollo de componentes, aunque hasta ahora todos los componentes habían sido stateless. De todas las maneras de crear componentes nos decantaremos en preferencia por las clases ES6, en detrimento del método `createClass()`. Si quieres saber más sobre este punto, lee primero el artículo [Componentes React mediante clases ES6](#).



## Componentes con estado / sin estado

Para comenzar queremos analizar estos dos conceptos en contraposición. Cuando empezamos con React pueden provocar dudas, pero realmente es sencillo. Básicamente, la diferencia entre componentes con o sin estado estriba en que los componentes con estado permiten mantener datos propios a lo largo del tiempo e implementar comportamientos en sus diferentes métodos del ciclo de vida.

Los componentes sin estado no guardan ninguna información y por ello no necesitan de datos locales. Todos los componentes implementados hasta el momento eran stateless, sin estado. Eso no significa que no puedan personalizar su comportamiento, lo que se consigue con la ayuda de las [propiedades de los componentes](#). Estas propiedades que nos pasan se podrán incluso transformar al producir una salida, de modo que sea acorde con las necesidades, pero no se guardará ningún valor y el componente no tendrá un ciclo de vida.

**Nota:** El ciclo de vida de los componentes es algo que todavía no hemos abordado. Más adelante hablaremos de él, porque entenderlo es muy importante en React.

## Ejemplo de componente sin estado

Para que quede más claro, esto sería una implementación de componente sin estado que recibe un timestamp como propiedad y genera una vista donde ese instante se muestra convertido en una fecha en español.

```
import React from 'react';

export default function (props) {
  var date = new Date(parseInt(props.timestamp, 10));
  var fecha = date.getDate() + '/' + (date.getMonth() + 1) + '/' + date.getFullYear();
  return (
    <span>{fecha}</span>
  );
}
```

Este componente se podría usar importándolo y luego colocando la etiqueta correspondiente. Me parece interesante mostrar cómo se usaría porque el componente en sí, tal como se ha definido, no tiene nombre. El nombre se lo asignamos a la hora de importarlo.

```
import React, { Component } from 'react'
import TimestampToDate from './utils/TimestampToDate'

class App extends Component {
  render() {
    return (
      <TimestampToDate timestamp={1475700297974} />
    );
  }
}
```

Otro detalle interesante aquí es la forma como le pasamos un valor numérico a un componente. Dentro del componente la propiedad timestamp se esperaría que fuera un entero y para que así sea tenemos que indicarle el valor sin comillas. Lo metemos entre llaves porque si no le colocas comillas JSX te obliga a que sea una expresión Javascript. Esa expresión se evalúa como un número entero.

Con este ejemplo queremos dejar claro que las propiedades de los componentes se pueden manipular para transformarlas en cualquier otro dato. Pero atención, las propiedades en sí debemos tratarlas como valores de solo lectura, algo que se hace así para evitar posibles situaciones inesperadas. Si queremos manipular las propiedades y transformarlas en otra cosa lo más normal es guardemos los datos nuevos generados como variables, o estado si fuera

necesario. Incluso, si solo se trata de una transformación sencilla para visualizar en la vista, podrías incluirla como una expresión Javascript embebida entre llaves dentro del JSX, pero por claridad del código es preferible crear variables locales para generar esos cambios, como en el ejemplo anterior. En todo caso, quédate con que las propiedades deben trabajar como solo lectura.

Todo eso en cuanto a componentes sin estado. Ahora pensemos en otro componente, en el necesitaríamos estado. Un ejemplo lo tendríamos en un contador. Ese contador puede incrementarse o decrementarse. Incluso podrían pasarnos como propiedad el valor inicial del contador, pero el valor actual de la cuenta lo guardaremos en el estado. Otro ejemplo podría ser un componente que se conecte con un API Rest. A este componente le podemos pasar como propiedad la URL del API y una serie de parámetros para realizar la solicitud al servidor. Una vez que recibamos los datos los común será almacenarlos como estado del componente, para usarlos como sea necesario.

El componente podrá además reaccionar al cambio de estado, de modo que actualice su vista cuando sea necesario. Eso lo veremos cuando analicemos el ciclo de vida de los componentes.

## Ejemplo de componente con estado

Seguro que estarás deseando ver ya un componente con estado. Enseguida nos ponemos con ello, pero quiero que veamos antes el código de "boilerplate" para crear un componente con estado.

```
import React from 'react'

export default class Contador extends React.Component {
  constructor(...args) {
    super(...args)
  }

  render() {
    return (
      <div>Esto aun no tiene estado!</div>
    )
  }
}
```

Lo primero decir que este componente todavía no tiene un estado implementado, es solo un código de partida para ver un par de cosas.

Como primer detalle no estamos importando implícitamente la clase Component, para hacer el extends. No es problema porque depende de React. Así que ahora estamos haciendo "extends React.Component", lo que es perfectamente válido. Esto no tiene mucho que ver con el tema que nos ocupa, de los estados, pero así vemos más variantes de codificación.

Ya más importante es el constructor. En este boilerplate la verdad es que el constructor no sirve para nada, porque realmente no hemos inicializado nada (como sabes, los constructores resumen las tareas de inicialización de los objetos). En este caso simplemente estamos

llamando al constructor de la clase padre, `super()`, pasándole los mismos argumentos que nos pasaron a nosotros.

**Nota:** Eso de "...args" es una desestructuración, algo que nos viene de ES6, que permite en este caso recibir todos los argumentos o parámetros enviados a una función, sin necesidad de indicarlos uno a uno.

Generalmente en los componentes que tienen estado necesitamos inicializar el estado, por lo que el sitio más correcto sería el constructor. Si realizamos un constructor tenemos que asegurarnos que se invoque al constructor de la clase padre, que realiza una serie de tareas. Como esa invocación hay que hacerla explícita al sobrescribir el constructor, nos obliga a escribir la llamada a `super()`. Enseguida veremos cómo inicializar el estado, pero he querido mostrar ese código, aún sin la inicialización, para comentar este detalle del constructor.

En el boilerplate encuentras también el método `render()`, que ya sabemos que es el que nos sirve para definir la representación del componente.

Ahora vamos ya con el estado. En nuestro ejemplo vamos a implementar el típico del contador. El valor actual del contador será nuestro estado y lo tendremos que inicializar en el constructor.

```
import React from 'react'

export default class Contador extends React.Component {
  constructor(...args) {
    super(...args)
    this.state = {
      contador: 0
    }
  }

  render() {
    return (
      <div>Cuenta actual: {this.state.contador}</div>
    )
  }
}
```

Ahora nuestro constructor ya tiene sentido, porque está realizando la inicialización de la propiedad "state" del componente. Como puedes ver el estado es un objeto, en el que ponemos tantos atributos como sea necesarios guardar como estado.

A la hora de renderizar el componente, por supuesto, podremos usar el estado para mostrar la salida. En este caso puedes ver cómo se vuelca el estado en la vista, con la expresión `{this.state.contador}`. Algo muy parecido a lo que hacíamos con las propiedades, solo que ahora los datos nos llegan del estado.



Solo nos falta implementar un botón para incrementar ese contador para ello tenemos que entrar en un tema nuevo, que son los eventos. Veamos el siguiente código.

```
import React from 'react'

export default class Contador extends React.Component {
  constructor(...args) {
    super(...args)
    this.state = {
      contador: 0
    }
  }

  incrementar() {
    this.setState({
      contador: this.state.contador + 1
    })
  }

  render() {
    return (
      <div>
        <span>Cuenta actual: {this.state.contador}</span>
        <button onClick={this.incrementar.bind(this)}></button>
      </div>
    )
  }
}
```

Ahora tenemos un botón y al hacer clic sobre él se invocará a la función `incrementar()`. Sin querer entrar en demasiados detalles sobre eventos, pues no es el asunto de este artículo, cabe decir que se pueden definir como si fueran atributos de los elementos o componentes. Como valor le colocamos su manejador de eventos. Además el `"bind(this)"` lo hacemos para bindear el contexto. Sobre eventos hablaremos más adelante.

Lo que es interesante, para lo que respecta al estado, está en el manejador de evento `incrementar()`. Este usa el método `setState()` para modificar el estado. El detalle que no se te puede escapar es que, para manipular el estado no se debe modificar "a pelo" (a mano) la propiedad `this.state`, sino que tenemos que hacerlo a través de `this.setState()`. El motivo es que `setState()`, además de alterar el estado, desencadena toda una serie de acciones implementadas en el core de React, que se encargan de realizar todo el trabajo por debajo para que ese cambio de estado tenga una representación en la vista.

Dicho de otra manera, en el momento que cambiemos el estado con `setState()`, se pone en ejecución el motor de React para que se actualice el DOM virtual, se compare con el DOM del navegador y por último se actualicen aquellos elementos que sea necesario (porque realmente hayan cambiado). Si no usamos `setState()` todas esas operativas no se producirían y los componentes empezarían a funcionar de manera no deseada.

## Conclusión

De momento hemos dado bastante información sobre el estado de los componentes, hemos podido distinguir componentes con estado y componentes sin estado, y aunque todavía nos quedan cosas que aprender, hemos podido ver ejemplos para hacernos una idea de cada tipo de componente,

En futuros artículos tenemos que abordar otros asuntos clave, como la inicialización de propiedades y estado ante diversos estilos de codificación, así como el ciclo de vida de los componentes, que es necesario conocer a fondo para implementar muchas de las necesidades de desarrollo en React.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 14/12/2016  
Disponible online en <http://desarrolloweb.com/articulos/estado-componentes-react.html>

# Propiedades y estado predeterminado en componentes React

**Cómo definir predeterminados para las propiedades de los componentes React, así como para su estado.**

En capítulos anteriores del [Manual de React](#) tuvimos la ocasión de aprender a crear, y usar, propiedades en los componentes. También analizamos el estado de los componentes y aprendimos a crear componentes statefull, con sus métodos para definir y alterar el estado, así como representar las vistas dependiendo del estado.

En este capítulo vamos a analizar los modos mediante los cuales podemos definir tanto los valores predeterminados de las propiedades, como del estado. Nos detenemos en este punto porque, dependiendo de cómo se creen los componentes, el modo de trabajo será diferente.

Antes de comenzar cabe simplemente recordar las variantes básicas para la creación de componentes, que ya conoces:

- Crearlos con el [método React.createClass\(\)](#)
- Crearlos mediante [clases de Javascript ES6\(\)](#)



## Primer render

Estas operaciones de carga de valores predeterminados en propiedades y estado se realizan guiados por unos métodos específicos de React, que se ejecutan durante la primera renderización del componente, justo antes de representarse en la página.

Como primer render nos referimos a la primera vez que se representa el componente dentro de la página, y se inyecta en el DOM del navegador. Cuando ocurre esa operación de renderizado, el componente debe tener tanto sus propiedades como su estado inicializados.

**Nota:** Estos métodos están muy ligados al ciclo de vida de los componentes, puesto que se ejecutan en etapas muy concretas de la vida de un componente. Analizaremos con detalle el ciclo de vida en el próximo artículo.

## Métodos para definir propiedades y estado predeterminado

Si queremos indicar tanto propiedades como estado con valores predeterminados debemos usar los siguientes métodos.

Vamos a comenzar viendo cómo se usan los métodos de renderizado con la variante de creación de componentes a partir de `React.createClass()`. Son dos métodos y los vamos a ver por orden de ejecución.

### `getDefaultProps()`

Este método se ejecuta primero y sirve para generar valores por defecto para las propiedades de un tipo de componente. Con tipo de componente nos referimos a todas las instancias que un componente puede generar. Por ejemplo, si un componente se usa 5 veces, este método se ejecutará una única vez, antes de comenzar a instanciar elementos concretos de ese componente.

Este método debe devolver un objeto, con el conjunto de propiedades cuyos valores predeterminados se quiera definir.

Los valores predeterminados en las propiedades funcionan de la siguiente manera.

Si al usar el componente se indican los valores de las propiedades, se toman esos valores dentro de la instancia del componente. No se indicaron los valores de esas propiedades al usar el componente, se crearán igualmente esas propiedades, cargando sus valores definidos mediante `getDefaultProps()`.

Este método solo es posible usarlo cuando estamos trabajando con la variante de creación de componentes `React.createClass()`.

### `getInitialState()`

Este método se ejecuta después de haberse definido las propiedades por defecto. Sirve para inicializar el estado inicial de un componente y solamente funciona en componentes creados mediante `React.createClass()`. A diferencia de `getDefaultProps()`, este método se ejecuta una vez por instancia del componente.

El objeto que devuelva `getInitialState()` será lo que se inicialice como estado del componente.

Veremos ahora un ejemplo sobre cómo se podrían realizar estas inicializaciones mediante el método de creación de componentes de `React.createClass()`.

```
var ComponenteInicializacion = React.createClass({
  getDefaultProps: function() {
    console.log('Ejecuto getDefaultProps')
    return {
      propiedad: 'Valor por defecto de una propiedad'
    }
  },

  getInitialState: function() {
    console.log('Ejecuto getInitialState');
    return {
      estado: 'inicializado en getInitialState'
    }
  },

  render: function() {
    console.log('Ejecuto render')
    return (
      <div>
        <p>Componente con propiedades y estado inicializado</p>
        <p>Estado: {this.state.estado}</p>
        <p>Propiedad: {this.props.propiedad}</p>
      </div>
    )
  }
})
```

Si usas este componente varias veces observarás gracias a los `console.log` que primero se inicializan las propiedades por defecto, una sola vez, y luego se inicializa el estado y se renderiza tantas veces como componentes hayas generado.

```
var anclaje = document.getElementById('anclaje')
ReactDOM.render(
  <div>
    <ComponenteInicializacion />
    <ComponenteInicializacion />
  </div>
  , anclaje)
```

Ahora te dejamos un archivo html completo donde podrás ver el componente en su declaración y funcionamiento, que podrás usar en un entorno vacío, sin necesidad de configurar transpiladores. Recuerda que esta alternativa no es la correcta para usar React en un entorno de producción, como se explicó en el artículo [Comenzar con ReactJS desde un entorno de desarrollo vacío](#).

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
<meta charset="UTF-8">
<title>Inicializar componentes con createClass</title>
</head>
<body>
  <div id="anclaje"></div>

  <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
  <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></script>
  <script src="https://unpkg.com/babel-core@5.8.38/browser.min.js"></script>

  <script type="text/babel">
    var ComponenteInicializacion = React.createClass({
      getDefaultProps: function() {
        console.log('Ejecuto getDefaultProps')
        return {
          propiedad: 'Valor por defecto de una propiedad'
        }
      },

      getInitialState: function() {
        console.log('Ejecuto getInitialState');
        return {
          estado: 'inicializado en getInitialState'
        }
      },

      render: function() {
        console.log('Ejecuto render')
        return (
          <div>
            <p>Componente con propiedades y estado inicializado</p>
            <p>Estado: {this.state.estado}</p>
            <p>Propiedad: {this.props.propiedad}</p>
          </div>
        )
      }
    })

    var anclaje = document.getElementById('anclaje')
    ReactDOM.render(
      <div>
        <ComponenteInicializacion />
        <ComponenteInicializacion />
      </div>
      , anclaje)

  </script>

</body>
</html>
```

Sirva entonces el anterior código solo para motivos de pruebas. Se podría ejecutar perfectamente haciendo doble clic sobre el archivo HTML, sin necesidad de un servidor web.

Ten en cuenta que nuestro método preferido es el siguiente, en el que realizaremos lo mismo desde un entorno completo con Create React App y clases ES6.

## Mecanismo de inicialización de propiedades y estado con clases ES6

Cuando estamos trabajando con clases ES6 estos mecanismos varían un poco. Los métodos `getDefaultProps()` y `getInitialState()` no te funcionarán. Si los intentas usar observarás mensajes del compilador y en la consola.

En el caso de la inicialización de propiedades, en componentes creados con clases de ES6, se debe realizar definiendo una propiedad "defaultProps" en la clase que implementa el componente. En el caso de la inicialización del estado, la operativa se realiza en el constructor. Se verá claramente con un ejemplo enseguida.

**Nota:** En el caso de la inicialización de propiedades, como se realiza una única vez para todos los componentes de un tipo, no tiene sentido realizarlo en el constructor, puesto que en ese caso se realizaría para cada una de sus instancias. Por tanto, en el constructor ya estarán inicializadas las propiedades con sus valores por defecto, o aquellos definidos al usar el componente.

En la alternativa de creación de componentes mediante clases ES6 el código que tendríamos que generar para la inicialización de las propiedades y estado sería el siguiente:

```
import React from 'react'

export default class CicloVida extends React.Component {
  constructor(...args) {
    console.log('Ejecuto constructor', ...args)
    super(...args)
    this.state = {
      estado: 'Inicializado en el constructor'
    }
  }
}

render() {
  return (
    <div>
      <p>Componente con propiedades y estado inicializado</p>
      <p>Estado: {this.state.estado}</p>
      <p>Propiedad: {this.props.propiedad}</p>
    </div>
  )
}

CicloVida.defaultProps = {
  propiedad: 'Valor por defecto definido para la propiedad'
}
```

Como estarás observando, la inicialización de propiedades se hace una vez definida la clase, mediante una propiedad que usamos dentro de la propia clase. Al no estar en el código de la clase, se ejecutará una única vez.

Ya para la inicialización de estado, se define en el constructor, con lo que se ejecutará para cada componente instanciado de manera independiente. Esto lo vimos con detalle en el artículo sobre el [estado de los componentes](#).

Continuaremos este manual con el ciclo de vida de los componentes, algo cuyas primeras trazas ya hemos empezado a ilustrar con este artículo.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 29/12/2016  
Disponible online en <http://desarrolloweb.com/articulos/propiedades-estado-predeterminado-componentes-react.html>



# Ciclo de vida de los componentes React

**Qué es el ciclo de vida y cómo implementar métodos dentro del ciclo de vida, para qué necesitamos usarlo.**

El ciclo de vida no es más que una serie de estados por los cuales pasa todo componente a lo largo de su existencia. Esos estados tienen correspondencia en diversos métodos, que nosotros podemos implementar para realizar acciones cuando se van produciendo.

En React es fundamental el ciclo de vida, porque hay determinadas acciones que debemos necesariamente realizar en el momento correcto de ese ciclo. Ese es el motivo por el que hay que aprenderse muy bien cuáles son las distintas etapas por las que pasa la ejecución de un componente React. No es un tema trivial y tiene diversos matices. Veremos una introducción en este artículo y pondremos ejemplos.

Antes de comenzar cabe decir que esto es algo específico de los componentes con estado, ya que los componentes sin estado tienen apenas un método que se usará para renderizar el componente y React no controlará su ciclo de vida a través de los métodos que veremos a continuación. Montaje, actualización y desmontaje.



Estas son las tres clasificaciones de los estados dentro de un ciclo de vida del componente.

- El montaje se produce la primera vez que un componente va a generarse, incluyéndose en el DOM.
- La actualización se produce cuando el componente ya generado se está actualizando.
- El desmontaje se produce cuando el componente se elimina del DOM.

Además, dependiendo del estado actual de un componente y lo que está ocurriendo con él, se producirán grupos diferentes de etapas del ciclo de vida. En la siguiente imagen puedes ver un resumen de esta diferenciación. Observarás que en el primer renderizado de un componente se pasa por etapas del ciclo de vida diferentes de las que se pasa cuando se produce un cambio en sus propiedades o un cambio en el estado, o su desmontaje.

Primer renderizado	Cambios en las propiedades	Cambio en el estado	Componente se desmonta
getDefaultProps *	componentWillReceiveProps	shouldComponentUpdate	componentWillUnmount
getInitialState *	shouldComponentUpdate	componentWillUpdate	
componentWillMount	componentWillUpdate	render *	
render *	render *	componentDidUpdate	
componentDidMount	componentDidUpdate		

La secuencia de ejecución de las etapas del ciclo de vida se da de arriba hacia abajo. Los elementos marcados con un asterisco naranja no son métodos del ciclo de vida propiamente dicho, pero sí que son etapas de ejecución de componentes y se producen en la secuencia descrita en la imagen.

**Nota:** En el [Manual de React](#) ya hemos visto todos los métodos marcados con el asterisco, por lo que deberían sonarte. Si no es así, simplemente revisa los artículos anteriores.

## Descripción de los métodos del ciclo de vida

Ahora vamos a describir las distintas etapas del ciclo de vida de los componentes y luego podremos ver ejemplos.

**Nota:** Puedes entender todos estos métodos como "hooks", ganchos que nos permiten ejecutar código en diferentes instantes. También los puedes ver como "manejadores de eventos", funciones con código a ejecutar cuando se producen diversas situaciones.

### componentWillMount()

Este método del ciclo de vida es de tipo montaje. Se ejecuta justo antes del primer renderizado del componente. Si dentro de este método seteas el estado del componente con `setState()`, el primer renderizado mostrará ya el dato actualizado y sólo se renderizará una vez el componente.

### componentDidMount()

Método de montaje, que solo se ejecuta en el lado del cliente. Se produce inmediatamente después del primer renderizado. Una vez se invoca este método ya están disponibles los elementos asociados al componente en el DOM. Si el elemento tiene otros componentes hijo, se tiene certeza que éstos se han inicializado también y han pasado por los correspondientes `componentDidMount`. Si se tiene que realizar llamadas Ajax, `setIntervals`, y cosas similares, éste es el sitio adecuado.

### componentWillReceiveProps(nextProps)

Método de actualización que se invoca cuando las propiedades se van a actualizar, aunque no en el primer renderizado del componente, así que no se invocará antes de inicializar las propiedades por primera vez. Tiene como particularidad que recibe el valor futuro del objeto de propiedades que se va a tener. El valor anterior es el que está todavía en el componente, pues este método se invocará antes de que esos cambios se hayan producido.

### **shouldComponentUpdate(nextProps, nextState)**

Es un método de actualización y tiene una particularidad especial con respecto a otros métodos del ciclo de vida, que consiste en que debe devolver un valor booleano. Si devuelve verdadero quiere decir que el componente se debe renderizar de nuevo y si recibe falso quiere decir que el componente no se debe de renderizar de nuevo. Se invocará tanto cuando se producen cambios de propiedades o cambios de estado y es una oportunidad de comprobar si esos cambios deberían producir una actualización en la vista del componente. Por defecto (si no se definen) devuelve siempre true, para que los componentes se actualicen ante cualquier cambio de propiedades o estado. El motivo de su existencia es la optimización, puesto que el proceso de renderizado tiene un coste y podemos evitarlo si realmente no es necesario de realizar. Este método recibe dos parámetros con el conjunto de propiedades y estado futuro.

### **componentWillUpdate(nextProps, nextState)**

Este método de actualización se invocará justo antes de que el componente vaya a actualizar su vista. Es indicado para tareas de preparación de esa inminente renderización causada por una actualización de propiedades o estado.

### **componentDidUpdate(prevProps, prevState)**

Método de actualización que se ejecuta justamente después de haberse producido la actualización del componente. En este paso los cambios ya están trasladados al DOM del navegador, así que podríamos operar con el DOM para hacer nuevos cambios. Como parámetros en este caso recibes el valor anterior de las propiedades y el estado.

### **componentWillUnmount()**

Este es el único método de desmontado y se ejecuta en el momento que el componente se va a retirar del DOM. Este método es muy importante, porque es el momento en el que se debe realizar una limpieza de cualquier cosa que tuviese el componente y que no deba seguir existiendo cuando se retire de la página. Por ejemplo, temporizadores o manejadores de eventos que se hayan generado sobre partes del navegador que no dependen de este componente.

## **Ejemplo de componente definiendo su ciclo de vida**

---

Vamos a hacer un ejemplo muy básico sobre estos métodos, que simplemente lanza mensajes a la consola cuando ocurren cosas. El objetivo es que podamos apreciar el orden de ejecución de todos estos métodos del ciclo de vida.

El código de este componente con los métodos del ciclo de vida implementados lo encuentras a

## continuación:

```
import React from 'react'

export default class CicloVida extends React.Component {
  constructor(...args) {
    console.log('Ejecuto constructor', ...args)
    super(...args)
    this.state = {
      estado: 'Inicializado en el constructor'
    }
  }

  componentWillMount() {
    console.log('Se ejecuta componentWillMount')
  }

  componentDidMount() {
    console.log('Se ejecuta componentDidMount')
  }

  componentWillReceiveProps(nextProps) {
    console.log('Se ejecuta componentWillReceiveProps con las propiedades futuras', nextProps)
  }

  shouldComponentUpdate(nextProps, nextState) {
    console.log('Ejecutando shouldComponentUpdate. Próximas propiedades y estado: ', nextProps, nextState)
    // debo devolver un booleano
    return true
  }

  componentWillUpdate(nextProps, nextState) {
    console.log('Ejecutando componentWillUpdate. Próximas propiedades y estado: ', nextProps, nextState)
  }

  componentDidUpdate(prevProps, prevState) {
    console.log('Ejecutando componentDidUpdate. Anteriores propiedades y estado: ', prevProps, prevState)
  }

  componentWillUnmount() {
    console.log('Se desmonta el componente...')
  }

  render() {
    return (
      <div>
        <p>Componente del ciclo de vida</p>
        <p>Estado: {this.state.estado}</p>
        <p>Propiedad: {this.props.propiedad}</p>
      </div>
    )
  }
}
```

```
CicloVida.defaultProps = {  
  propiedad: 'Valor por defecto definido para la propiedad'  
}
```

Solo con que uses este componente tal cual podrás ver ya cómo se ejecutan los métodos del ciclo de vida relacionados con la parte del "primer renderizado" (mira la imagen anterior de este artículo).

Si quisieras poner en marcha otros métodos del ciclo de vida necesitarías hacer cosas como cambiar el estado del componente o sus propiedades. Como referencia te dejamos este otro código, en el que usamos el componente de CicloVida y cambiamos de manera dinámica una de sus propiedades.

```
import React from 'react'  
import CicloVida from './index'  
  
export default class UsarCicloVida extends React.Component {  
  constructor(...args) {  
    super(...args)  
    this.state = {  
      valorPropiedad: 'Test de valor de propiedad'  
    }  
  }  
  
  cambiarPropiedad() {  
    this.setState({  
      valorPropiedad: 'Otro valor'  
    })  
  }  
  
  render() {  
    return (  
      <div>  
        <button onClick={this.cambiarPropiedad.bind(this)}>Cambiar propiedad</button>  
        <CicloVida propiedad={this.state.valorPropiedad} />  
      </div>  
    )  
  }  
}
```

Ese código hace uso del componente anterior. Lo podemos analizar parcialmente, puesto que todavía hay cosas aquí que no hemos llegado a tocar. Pero lo básico en que fijarse es en la propiedad bindeada. El estado del componente "UsarCicloVida" tiene un dato que se ha bindeado como valor de la propiedad del componente "CicloVida". Por otra parte tenemos un botón que cambia el estado, y por consiguiente el cambio de estado se traduce en una actualización de la propiedad del componente CicloVida, lo que desencadenará en la ejecución de los métodos del lifecycle relacionados con el cambio del valor de una propiedad.

Seguramente que, si tienes alguna experiencia con el desarrollo de librerías o frameworks que usen binding, podrás entender el ejemplo. Pero, como decíamos, son cosas que tenemos que

ver con calma en futuros artículos de este manual.

De momento lo dejamos por aquí. Tienes bastante material para experimentar. Haremos otros ejemplos del ciclo de vida más interesantes en futuros artículos.

Este artículo es obra de *Miguel Angel Alvarez*  
Fue publicado por primera vez en 17/01/2017  
Disponible online en <http://desarrolloweb.com/articulos/ciclo-vida-componentes-react.html>